Rose-Hulman Undergraduate Mathematics Journal

Volume 7
Issue 2

Article 2

Formulas for Computable and Noncomputable Functions

Samuel Alexander
University of Arizona, alexander@math.ohio-state.edu

Formulas for Computable and Noncomputable Functions

Samuel Alexander
The University of Arizona
With supervision of Dr. Ksenija Simic
The University of Arizona

ABSTRACT

We explore the problem of writing explicit formulas for integer functions. We demonstrate that this can be done using elementary machinery for a wide class of functions. Constructive methods are given for obtaining formulas for computable functions and for functions in the arithmetical hierarchy. We include a short background on computability theory.

1 Introduction

A common problem in the amateur literature reads: "find a formula for (some function)." We attack this problem in very broad generality by solving it when the function in question is computable or within the arithmetical hierarchy.

But the first step toward solving this problem is deciding what precisely is meant by a formula. There is no standard definition for a formula, so before we can proceed to the problem, we must make quite clear what we mean. Loosely speaking, we allow formulas using standard arithmetic, finite sums, the Kronecker δ function (a very simple function defined below), and infinite series.

First, we offer some background on computability theory.

2 Background

Throughout math and science, it is common to find concepts and notions that are useful, or indeed indispensable, but which are not really well understood for a long time. Eventually as these notions see more use, they are formulated with greater precision, and it is precisely at this point when huge leaps of progress are often made. One concept which has been absolutely fundamental since ancient times is the *algorithm*: a set of instructions explaining how some task can be accomplished. And yet, as universal and powerful as this notion is, it was

not made formal until the first half of the 20^{th} century. Thus, we should not be surprised that when the notion finally was set on firm ground, the breakthroughs that ensued were stunning and numerous.

Since its conception at the hands of mathematicians like Alan Turing and Alonzo Church, computability theory– essentially, the abstract study of algorithms and effectively computable tasks– has blossomed into a strong field whose applicability throughout diverse branches of mathematics is paralleled only by its own deep beauty.

What set all this in motion was the revelation that the many seemingly disparate attempts to define an effectively computable function– that is, a function which, informally, can be calculated using an algorithm– were in fact all precisely equivalent. For example, Turing suggested the popular Turing machine definition, where computable functions are those which can, in a formal sense, be programmed into a mathematical model of a computer– a Turing machine. Contrast this with Gödel's recursive functions (defined below), a very formal and abstract construction that bears no outward resemblance to Turing computable functions; and contrast either of these with Church's λ-computable functions, essentially functions that can be given by some very specific formulas (for Turing computability and recursive functions, see [2], for λ-calculus see [1]). Astonishingly, these separate attempts to contrive definitions of effectively computable functions, turn out to give exactly the same functions! That is to say, a function is Turing machine computable if and only if it is recursive if and only if it is λ-computable. This allowed Church and Turing to lay the foundation of computability theory with the *Church-Turing Thesis*, which basically states: the various attempts to define effectively computable functions are successful, that is to say, functions that are effectively computable are identical to those given by the various attempts to define computable functions.

It must be emphasized that the Church-Turing thesis is *not* a formal mathematical statement. It is an informal statement about "effectively computable functions". The Church-Turing thesis cannot be proved or disproved, because "effectively computable" is itself not a formal mathematical term, but rather an informal word for those functions for which an algorithm exists. It is more accurate to call the thesis a philosophical statement than a mathematical statement. Its use in mathematics is not formal: rather, it is a kind of labor-saving device, as the following example shows.

Example: Suppose we want to prove that the function which gives the nth prime number is computable, in the sense of some given model of computation, say Turing machine computable. To do so formally requires almost no ingenuity (at least once the basic techniques of programming with Turing machines are understood), but involves quite a lot of tedious busywork. On the other hand, informally we *know* the nth prime number function can be computed via an algorithm, for example the Sieve of Eratosthenes; the Church-Turing thesis states, informally, that the various definitions of computable functions– including Turing's definition– include precisely the functions which are effectively computable. Thus, we can write: "The nth prime number can be calculated by an unam-

biguous algorithm in a finite amount of time, so by the Church-Turing thesis, f is Turing computable". This is *not* a rigorous mathematical proof, but it is accepted as proof in the literature, with the implicit understanding that if a critic would challenge it, it would be only a simple (but tedious and long) exercise to produce an actual proof.

Having said all this, we should actually give one of the definitions of computable function. Although they are all equivalent, the different definitions lend themselves more readily to different applications– another very practical feature of computability theory. The definition that will be most useful to our own work is the recursive function definition of Gödel.

When we write $f :\subseteq A \to B$, we mean that f is a function whose domain is a subset of A (possibly A itself). In much of mathematics it is a convention to explicitly state exactly a function's domain, but this convention would rapidly become a huge pointless eyesore in computability theory, where the domain of a function is often a bizarre and not-really-important set of tuples of natural numbers.

Definition 1: The *computable* functions (also called the *recursive* functions) are defined inductively as follows:

1. The *basic* functions are computable. These are the *zero function* $O : \mathbb{N} \to \mathbb{N}$ defined by $O(n) = 0$; the *successor function* $\sigma : \mathbb{N} \to \mathbb{N}$ defined by $\sigma(n) = n + 1$; and the *projection functions* defined, for all $i, k \in \mathbb{N}$, $0 < i \leq k$, by

$$\pi_i^k : \mathbb{N}^k \to \mathbb{N} : (n_1, ..., n_k) \mapsto n_i.$$

2. (Closure under Composition) If $\psi :\subseteq \mathbb{N}^k \to \mathbb{N}$ is computable and for $1 \leq i \leq k$, $\varphi_i :\subseteq \mathbb{N}^m \to \mathbb{N}$ is computable, then the function $f :\subseteq \mathbb{N}^m \to \mathbb{N}$ defined by
$$f(\vec{n}) = \psi(\varphi_1(\vec{n}), ..., \varphi_k(\vec{n}))$$
is also computable. Here we write $\vec{n}$ for $n_1, ..., n_m$; this will be done frequently throughout the paper, and should cause no confusion.

3. (Closure under Recursion) If $g :\subseteq \mathbb{N}^k \to \mathbb{N}$ is computable and $h :\subseteq \mathbb{N}^{k+2} \to \mathbb{N}$ is computable, then the function $f :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ defined by the recurrence relation
$$f(\vec{n}, m) = \begin{cases} g(\vec{n}) \text{ if } m = 0, \\ h(\vec{n}, f(\vec{n}, m-1), m-1) \text{ otherwise} \end{cases}$$
is computable.

4. (Closure under Unbounded Minimization) Suppose $g :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ is computable. Define the function $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ by
$$f(x_1, ..., x_k) = \min\{y | g(x_1, ..., x_k, y) = 0 \text{ and } g(x_1, ..., x_k, z) \text{ exists } \forall z < y\},$$
whenever any such y exists. Then f is computable.

In the appendix the reader can find some explicit proofs that some functions are recursive; if one is unfamiliar with the mechanics of inductive definitions like the one above, a leisurely glance through the appendix would no doubt shed light. The important thing is not the nitty-gritty details but rather the fact that the computable functions essentially capture the intuitive idea of functions that can be computed by some algorithm.

Note that closure under unbounded minimization allows functions whose domains are not all of $\mathbb{N}^k$ for any k, to still possibly be computable. It basically allows us to build a "brute force search" into a computable function– even if the brute force search will never succeed at some input (in which case the resulting function is undefined at that input). Closure under unbounded minimization basically allows us to accomodate brute force searches in our algorithms.

There is a subtle point here: we are only considering functions which accept at least one input, in the spirit of most mathematics. In this, we are following Bilaniuk in [2]. Many computability theorists allow so-called "0-place functions" which are basically constant. The difference between the two approaches is superficial: a 0-place function can always be emulated with a 1- or more-place function which simply ignores its inputs.

With this knowledge of computability theory, we can turn our attention toward the problem at hand: using computability theory to attack the formula problem in very broad generality.

3 Formalizing Formulas

We often speak of formulas in an intuitive sense. We might say that a certain function has a formula in terms of basic arithmetic and exponents, for example. How would we formally define the set of functions that have formulas in terms of basic arithmetic and exponents? The most straightforward way to pull this off would be an inductive definition: we begin by listing some basic functions as being in the set; for example, constant functions and the identity function. Then we would start listing some *closure properties*: for example, closure under addition. We would specify that if f and g are in our set, then $f + g$ is also in our set. We would specify similar closure properties for multiplication, division and exponents, and we would be done.

To prove that something like $f(x) = 5x^2+2^x+4$ is in our set, we would begin by noting that 5, 2, 4 and x are in our set, being constants and the identity function. Then x^2 and 2^x are in our set by closure under exponents, so $5x^2$ is in our set by closure under multiplication, and finally $5x^2 + 2^x + 4$ is in our set by closure under addition.

The type of functions we grant formula status to depends on what kind of formulas we're intuitively thinking of. If we're thinking of formulas that can involve infinite series, then the above example would not suffice. For this reason, there is no, and indeed can be no, standard definition of functions which have explicit formulas.

Now suppose $f(x, y)$ has a formula. Then $g(x, y) = f(y, x)$ should intuitively

have a formula, and so should $h(x,y,z) = f(x,y)$. But how can we take care of these in our definition? The answer is a combination of *composition* and *projection* (as defined in Definition 1). We include the projections π_i^k ($1 \leq i \leq k$) as having a formula in the base case of our inductive definition, and then we allow composition, so that, since $f(x,y)$ has a formula and π_1^2, π_2^2 have formulas, $f(\pi_2^2(x,y), \pi_1^2(x,y))$ has a formula. But this is just $f(y,x)$. And likewise $f(\pi_1^3(x,y,z), \pi_2^3(x,y,z))$ has a formula, which is to say $h(x,y,z) = f(x,y)$ has a formula. Note that by including the projections, we get the identity function π_1^1 as a special case, and so do not need to mention it explicitly.

With all this in mind, we give the definition we shall use throughout this discourse.

Definition 2: The set F of *functions that have explicit formulas* is defined inductively as follows.

1. F includes the *atomic formula functions*, which are: the constantly 0 and constantly 1 functions $O : \mathbb{N} \to \mathbb{N} : n \mapsto 0$ and $\mathbf{1} : \mathbb{N} \to \mathbb{N} : n \mapsto 1$; for each $k > 0$, $0 < i \leq k$, the projection $\pi_i^k : \mathbb{N}^k \to \mathbb{N} : (n_1, ..., n_k) \mapsto n_i$; the Kronecker δ function

$$\delta : \mathbb{N}^2 \to \mathbb{N} : (n,m) \mapsto \begin{cases} 1 \text{ if } m = n, \\ 0 \text{ otherwise;} \end{cases}$$

   and the functions

$$\begin{aligned} \mathbf{Add} &: \mathbb{N}^2 \to \mathbb{N} : (m,n) \mapsto m+n, \\ \mathbf{Multiply} &: \mathbb{N}^2 \to \mathbb{N} : (m,n) \mapsto mn, \\ \mathbf{Subtract} &: \mathbb{N}^2 \to \mathbb{N} : (m,n) \mapsto |m-n|, \\ \mathbf{Power} &: \mathbb{N}^2 \to \mathbb{N} : (m,n) \mapsto m^n \end{aligned}$$

   (we define 0^0 to be 1).

2. F is closed under function composition (which is defined as it was in part 2 of Definition 1).

3. F is closed under infinite series and finite series, that is to say, if $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ and $g :\subseteq \mathbb{N}^{k-1} \to \mathbb{N}$ are in F, $k > 1$, then the $(k-1)$-variable functions $\sum_{i=0}^{\infty} f(\vec{n}, i)$ and $\sum_{i=0}^{g(\vec{n})} f(\vec{n}, i)$ are in F. Note the domain of $\sum_{i=0}^{\infty} f(\vec{n}, i)$ only includes those $\vec{n}$ such that the series converges.

■

Note that since F contains **Add**, **Multiply**, **Subtract**, and **Power**, and since F is closed under composition, it follows F is closed under addition, multiplication, absolute value subtraction, and exponentiation of functions.

Example: Suppose $f(x,y,z) = 2x + \delta(x,z) + xyz^{|x-y|}$. Intuitively we can tell at a glance that f has a formula using just the machinery from Definition 2. But to rigorously prove it, we proceed thus:

1. $f_1(x,y,z) = 2x$ is in F because $2x = \pi_1^3(x,y,z) + \pi_1^3(x,y,z)$ (here we're using closure under addition).
2. $f_2(x,y,z) = \delta(x,z)$ is in F because $\delta(x,z) = \delta(\pi_1^3(x,y,z), \pi_3^3(x,y,z))$ (here we're using closure under composition).
3. $f_3(x,y,z) = xyz^{|x-y|}$ is in F because we can write it as

$$f_3(x,y,z) = \pi_1^3(x,y,z)\pi_2^3(x,y,z)\pi_3^3(x,y,z)^{|\pi_1^3(x,y,z)-\pi_2^3(x,y,z)|}$$

   (here we're using closure under basic arithmetic repeatedly).
4. $f(x) = 2x+\delta(x,z)+xyz^{|x-y|}$ is in F by the previous observations combined with closure under addition. ■

Unless it is not immediately obvious that a formula is in F, we will not explicitly hash out the proofs, and in practically all cases it will suffice to simply give a formula for a function, to establish it is in F.

We shall use $\overline{\delta}(x,y)$ to denote $|1-\delta(x,y)|$. Of course $\overline{\delta} \in F$. Just for the record we write

$$\overline{\delta}(x,y) = \begin{cases} 1 \text{ if } x \neq y \\ 0 \text{ otherwise.} \end{cases}$$

The following lemma will be used repeatedly throughout. It provides some heuristical guidance for crafting formulas.

Lemma 1: Suppose $f : \mathbb{N}^k \to \{0,1\}$.

1. Let $g : \mathbb{N}^k \to \mathbb{N}$ be defined by: $g(\vec{n},i)$ equals the number of 1's that occur in $f(\vec{n},0), f(\vec{n},1), ..., f(\vec{n},i)$. Then $g \in F$ if $f \in F$, and in any case

$$g(\vec{n},i) = \sum_{j=0}^{i} f(\vec{n},j).$$

2. Let $g : \mathbb{N}^{k-1} \to \mathbb{N}$ be defined by: $g(\vec{n}) = 1$ if $f(\vec{n},i) \neq 0$ for some $i \in \mathbb{N}$, otherwise $g(\vec{n}) = 0$. Then $g \in F$ if $f \in F$, and in any case

$$g(\vec{n}) \quad = \quad \sum_{i=0}^{\infty} \overline{\delta}(0, f(\vec{n},i))\delta\left(1, \sum_{j=0}^{i} \overline{\delta}(0, f(\vec{n},j))\right)$$

Proof:

1. Each ith term adds 1 to the overall sum if and only if $f(\vec{n},i) \neq 0$.
2. The ith term adds 1 to the overall sum precisely when $f(\vec{n},i) \neq 0$ and the number of j, $0 \leq j \leq i$, such that $f(\vec{n},j) \neq 0$, is precisely 1, i.e., i is minimal with this property.

■

Among F, the set of functions with formulas in the sense of Definition 2, we single out a very special subset.

Definition 3: By F_0, we mean those functions in F which can be proved to be in F without appealing to the closure of F under infinite summation.

Intuitively speaking, a function is in F_0 if and only if it has an explicit formula using only the machinery of Definition 2, and specifically *not* using infinite series.

The next function will prove enormously useful for proving that various functions have formulas in the sense of Definition 2.

Definition 4: The *prime exponent function* $p : \mathbb{N}^2 \to \mathbb{N}$ is defined thus: if $x = 0$ or $y = 0$ then $p(x, y) = 0$; otherwise, $p(x, y)$ is the power appearing on the yth prime in the prime factorization of x. We'll usually write $p_y(x)$ for $p(x, y)$.

Lemma 2: $p \in F_0$.

Loosely speaking, Lemma 2 says that p has an explicit formula using just the machinery of Definition 2, and without any infite summation.

Proof of Lemma 2: Define $f : \mathbb{N}^2 \to \{0, 1\}$ by

$$f(a, b) = \begin{cases} 1 \text{ if } a \neq 0,\ b \neq 0, \text{ and } a|b, \\ 0 \text{ otherwise.} \end{cases}$$

We claim

$$f(a, b) = \overline{\delta}(a, 0)\overline{\delta}(b, 0) \sum_{i=0}^{b} \delta(ai, b).$$

If $a = 0$ or $b = 0$ the righthand side is killed by $\overline{\delta}(a, 0)$ or $\overline{\delta}(b, 0)$, respectively, and the claim holds. Assume $a > 0$ and $b > 0$, so $\overline{\delta}(a, 0)\overline{\delta}(b, 0) = 1$. If $a|b$ there is a unique $0 \leq k \leq b$ such that $ak = b$. Then the kth term in the sum is 1 and all other terms are 0, so the whole righthand side is 1. If $a \nmid b$ then no such k exists and *every* term in the sum is 0, so the righthand side is 0 and the claim holds.

Next, define $g : \mathbb{N} \to \{0, 1\}$ by $g(n) = 1$ if n is prime, $g(n) = 0$ otherwise. Now, n is prime if and only if there are precisely two distinct i with $i|n$, that is, precisely if there are two distinct i with $f(i, n) = 1$. So by part 1 of Lemma 1,

$$g(n) = \delta\left(2, \sum_{i=0}^{n} f(i, n)\right).$$

Next, define $h : \mathbb{N} \to \mathbb{N}$ thus: $h(n)$ shall be the nth prime number when $n > 0$, and $h(0) = 0$. We claim

$$h(n) = \sum_{i=0}^{2^n} ig(i)\delta\left(n, \sum_{j=0}^{i} g(j)\right).$$

From part 1 of Lemma 1 it follows that

$$\delta\left(n, \sum_{j=0}^{i} g(j)\right)$$

is 1 or 0 depending whether or not there are precisely n prime numbers between 0 and i inclusive. If i is the nth prime number, then there are precisely n primes between 0 an i inclusive, so the ith term in the sum is $ig(i) = i$. Otherwise, if i is composite, then $g(i) = 0$ so the ith summand vanishes, and if i is prime but not the nth prime, then there are not precisely n primes between 0 and i inclusive, so again the ith summand vanishes. So the only term that can possibly be nonzero is the $h(n)$th term. This term is present because $h(n) \leq 2^n$ (any elementary number theory book can be consulted for a proof of this fact). So the one nonzero term is the $h(n)$th term, which equals $h(n)$, establishing the claim.

Now we claim for $x, y \in \mathbb{N}$ that

$$p(x,y) = \sum_{i=0}^{x} f\left(h(y)^{i+1}, x\right).$$

If $x = 0$ or $y = 0$ this is immediate. Suppose $x > 0$ and $y > 0$. Then $h(y)$ is the yth prime, and $f(h(y)^{i+1}, x)$ is 1 if and only if $h(y)^{i+1}|x$. So if m is the maximum integer with $h(y)^m|x$, then the $i = 0$, $i = 1$, ..., $i = m - 1$ terms will all equal 1 and any further terms will equal 0 (note that the sum has at least m terms because $x \geq h(y)^m > m$). This proves the claim.

We can expand this formula for $p(x, y)$ by using our formula for h, and we can expand the result using our formula for g, and we can expand that result using our formula for f. This gives a formula for $p(x, y)$ using just machinery of Definition 2 and using no infinite series. ■

Since infinite series allow for functions to diverge at some points, the following terminology is useful:

Definition 5: Let f be a function. By *dom* f, we mean the domain of f. If *dom* $f = \mathbb{N}^k$ for some $k > 0$, we say that f is a *total function.*

Note that functions in F_0 are total since the only way to cause a function to be undefined at some point using just macheriny from Definition 2 would be to take an infinite series at some point.

Remark: The wary reader might feel uncomfortable about our declaring $0^0 = 1$, since there is no standard definition of 0^0, many authors defining it to be 1, some defining it to be 0, and others leaving it undefined altogether. As a matter of fact, the functions which have or don't have formulas according to

Definition 2 do not depend on what stance is taken on 0^0. To see this, let

$$f(x,y) = \begin{cases} x^y \text{ if } x \neq 0 \text{ or } y \neq 0 \\ 1 \text{ otherwise,} \end{cases}$$

$$g(x,y) = \begin{cases} x^y \text{ if } x \neq 0 \text{ or } y \neq 0 \\ 0 \text{ otherwise,} \end{cases}$$

$$h(x,y) = \begin{cases} x^y \text{ if } x \neq 0 \text{ or } y \neq 0 \\ \text{undefined otherwise.} \end{cases}$$

Then by simply considering the cases $x = y = 0$ or not, we have

$$f(x,y) = g(x,y) + \delta(x,0)\delta(y,0)$$
$$g(x,y) = h(x, y + \delta(x,0))$$
$$h(x,y) = f(x,y) + \sum_{i=0}^{\infty} \delta(x,0)\delta(y,0),$$

where in the third equation both sides are undefined at $x = y = 0$. Thus, all other machinery of Definition 2 being fixed, it does not matter which version of exponent we allow, we will get the other two for free. We take $0^0 = 1$ because it will prove very convenient that functions in F_0 be total.

4 Formulas for Computable Functions

In this section we show that all recursive functions are in F and any given recursive function can be proved to be in F using at most one appeal to closure under infinite series. Intuitively what this means is that every recursive function has a very basic explicit formula using no more powerful machinery than possibly one infinite summation.

Definition 6: The following terminology will shed considerable intuitive light on the beautiful ideas buried under a heap of details in Lemma 6 below.

Let $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ and $\tilde{f} : \mathbb{N}^{k+1} \to \mathbb{N}$ be functions; note that $\tilde{f}$ is total, although f may not be. We say that $i \in \mathbb{N}$ is a *key* to f at $\vec{n}$ via $\tilde{f}$ if either

1. $\tilde{f}(\vec{n}, i) = 0$, or
2. $\vec{n} \in dom\, f$ and $\tilde{f}(\vec{n}, i) = f(\vec{n}) + 1$.

If the 2nd case applies (the two cases are clearly mutually disjoint) then we say i is a *proper key* to f at $\vec{n}$ via $\tilde{f}$.

This is summarized in the following table:

Scenario	Diagnosis of i
$\tilde{f}(\vec{n}, i) = 0$	i is a key but not a proper key (to f at $\vec{n}$ via $\tilde{f}$)
$\tilde{f}(\vec{n}, i) = f(\vec{n}) + 1$	i is a proper key (to f at $\vec{n}$ via $\tilde{f}$)
Any other case	i is not a key (to f at $\vec{n}$ via $\tilde{f}$)

Definition 7: We say $\tilde{f}$ is a *gate* to f if for all $\vec{n} \in \mathbb{N}^k$:

1. For all $i \in \mathbb{N}$, i is a key to f at $\vec{n}$ via $\tilde{f}$, and
2. If $\vec{n} \in dom\ f$ then there exists a proper key i to f at $\vec{n}$ via $\tilde{f}$.

The first requirement can be thought of as stating that $\tilde{f}$ "has a keyhole", into which any key whatsoever can be inserted, but which will not necessarily respond to all keys (some keys will not turn the lock). The second requirement can be thought of as stating that *some* key will turn the lock, at least where the domain of f allows as much.

An intuitive way of thinking of keys and gates is as follows. If we want to find a function f at a point $\vec{n}$ in its domain, and we have a gate $\tilde{f}$ to f whose values we know, we need only plug a proper key into $\tilde{f}$ and the value of f will appear (plus 1). The reason for adding 1 is so that we can tell when we've found a proper key– it will produce a nonzero value in $\tilde{f}$. Of course we can easily subtract the 1 back off.

At this point we will dispatch a number of lemmas. These are very technical, but beneath the technicalities is buried a notion of computation by guessing which is fundamental to the work. The lemmas will culminate in an alternate characterization for computable functions in terms of gates, keys, and formulas, and this will lend itself well to our attack on the formula problem for computable functions.

Throughout, $D : \mathbb{N} \to \mathbb{N}$ shall be defined by $D(n) = n - 1$ if $n > 0$, $D(0) = 0$; D is in F_0 because evidently $D(n) = ||n - 1| - \delta(0, n)|$ (the absolute values around $n-1$ are required because we have no machinery for dealing with negative numbers; when $n = 0$, $|n - 1| = 1$ while we want $D(n) = 0$, hence the $-\delta(0, n)$ term; the outer absolute value does not actually avoid any negative numbers, but is nonetheless necessary if one wants to follow Definition 2 to the letter).

Lemma 3: Let S be the set (for all k) of functions $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ such that there exists a gate $\tilde{f} : \mathbb{N}^{k+1} \to \mathbb{N}$ to f, such that $\tilde{f} \in F_0$ (in words, S is the set of functions of any number of variables with gates that have formulas that do not use infinite summation). Then S is closed under composition.

Proof: Suppose $\psi :\subseteq \mathbb{N}^r \to \mathbb{N}$ (so $r > 0$), $\phi_i :\subseteq \mathbb{N}^k \to \mathbb{N}$ $(i = 1, 2, ..., r)$ are in S. So there exist gates $\tilde{\psi} : \mathbb{N}^{r+1} \to \mathbb{N}$, $\tilde{\phi}_i : \mathbb{N}^{k+1} \to \mathbb{N}$, in F_0. Let $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ be defined by $f(\vec{n}) = \psi(\phi_1(\vec{n}), ..., \phi_r(\vec{n}))$. We must show f has a gate $\tilde{f} \in F_0$.

Define $\tilde{f} : \mathbb{N}^{k+1} \to \mathbb{N}$ by

$$\tilde{f}(\vec{n}, i) = \tilde{\psi}(D(\tilde{\phi_1}(\vec{n}, p_1(i))), ..., D(\tilde{\phi_r}(\vec{n}, p_r(i))), p_{r+1}(i)))\Omega(\vec{n}, i),$$

where

$$\Omega(\vec{n}, i) = \delta\left(0, \sum_{j=1}^{r} \delta\left(0, \tilde{\phi_j}(\vec{n}, p_j(i))\right)\right).$$

Since r is fixed, by applying closure under addition repeatedly[1] we see $\Omega \in F_0$, and then it's clear $\tilde{f} \in F_0$. We will show $\tilde{f}$ is a gate to f. First consider Ω.

[1] There is a subtle point here. It is tempting to say we can apply closure under finite sigma

When can it be zero? It is zero precisely when $\tilde{\phi}_j(\vec{n}, p_j(i)) = 0$ for some $1 \leq j \leq r$. Thus Ω is 0 precisely if $p_j(i)$ is not a proper key to $\tilde{\phi}_j$ for some j; otherwise it is 1. In the former case, $\tilde{f}(\vec{n}, i)$ becomes 0 and i is not a proper key. In the latter, the $p_j(i)$ *is* a proper key to $\tilde{\phi}_j$ for all j, so $\tilde{f}(\vec{n}, i) = \tilde{\psi}(\phi_1(\vec{n}), ..., \phi_r(\vec{n}), p_{r+1}(i))$. Thus $\tilde{f}(\vec{n}, i)$ is $f(\vec{n}) + 1$ precisely if $p_{r+1}(i)$ is a proper key to ψ and each $p_j(i)$ is a proper key to ϕ_j; and $\tilde{f}(\vec{n}, i) = 0$ otherwise. This demonstrates $\tilde{f}$ is a gate as claimed, and proves the lemma.

■

Observe how the function D defined above plays the role of subtracting off the 1 when we get an answer from the gate– the gate, by definition, returns a value 1 higher than we want, so as to allow identification of non-proper keys. We'll be using D in this way often in the following lemmas.

We'll give an example to show more clearly what is really going on in the construction of Lemma 3.

Example: Throughout this example, all functions will have the obvious domains and codomain $\mathbb{N}$. Define

$$\begin{aligned}
\psi(x, y) &= x + y \\
\tilde{\psi}(x, y, i) &= (x + y + 1)\delta(i, 5) \\
\phi_1(x, y, z) &= xy \\
\tilde{\phi}_1(x, y, z, i) &= (xy + 1)\delta(i, 7) \\
\phi_2(x, y, z) &= yz + 1 \\
\tilde{\phi}_2(x, y, z, i) &= (yz + 2)\delta(i, 9).
\end{aligned}$$

Then $\tilde{\psi}$, $\tilde{\phi}_1$, and $\tilde{\phi}_2$ are F_0 gates for ψ, ϕ_1, and ϕ_2 respectively: it's easy to see $\tilde{\psi}(x, y, i)$ is $\psi(x, y) + 1$ if $i = 5$, 0 otherwise; that $\tilde{\phi}_1(x, y, z, i)$ is $\phi_1(x, y, z) + 1$ if $i = 7$, 0 otherwise; and that $\tilde{\phi}_2(x, y, z, i)$ is $\phi_2(x, y, z) + 1$ if $i = 9$, 0 otherwise. The astute reader will note these gates are unnecessarily complicated, for instance $x + y + 1$ would be a simpler gate for ψ; however, by restricting the proper keys in this way, the inner workings of Lemma 3 will be more fully exposed. We'll use Lemma 3 to obtain an F_0 gate $\tilde{f}$ for

$$f(x, y, z) = \psi(\phi_1(x, y, z), \phi_2(x, y, z)) = xy + yz + 1.$$

notation here, but we cannot. To do so, we would need to know that the function

$$s(\vec{n}, i, j) = \delta\left(0, \tilde{\phi}_j(\vec{n}, p_j(i))\right)$$

had a formula, but "taking the jth subindex" is not one of our allowed operations (however, $p_j(i)$ is fine since, the reader recalls, it's just shorthand for $p(i, j)$). So instead we must use closure under addition $r - 1$ times. For example if $r = 1000$ then the formula for Ω will contain 1000 individual summands rather than a single slick sigma notation symbol. But since r is a fixed constant, this is fine.

Writing $\vec{x}$ for x, y, z, we define Ω by

$$\begin{aligned}\Omega(\vec{x}, i) &= \delta\left(0, \delta\left(0, \tilde{\phi}_1(\vec{x}, p_1(i))\right) + \left(0, \tilde{\phi}_2(\vec{x}, p_2(i))\right)\right) \\ &= \delta\left(0, \delta\left(0, (xy+1)\delta(p_1(i), 7)\right) + \delta\left(0, (yz+2)\delta(p_2(i), 9)\right)\right).\end{aligned}$$

If $p_1(i) \neq 7$ or $p_2(i) \neq 9$ then at least one of $(xy+1)\delta(p_1(i), 7)$ or $(yz+2)\delta(p_2(i), 9)$ will vanish, and we'll get $\Omega(\vec{x}, i) = 0$. Otherwise neither will vanish and we'll get $\Omega(\vec{x}, i) = 1$. So

$$\Omega(\vec{x}, i) = \begin{cases} 0 \text{ if } p_1(i) \neq 7 \text{ or } p_2(i) \neq 9 \\ 1 \text{ otherwise.} \end{cases}$$

Next, still following Lemma 3, we define

$$\begin{aligned}\tilde{f}(\vec{x}, i) &= \tilde{\psi}\left(D\left(\tilde{\phi}_1(\vec{x}, p_1(i))\right), D\left(\tilde{\phi}_2(\vec{x}, p_2(i))\right), p_3(i)\right)\Omega(\vec{x}, i) \\ &= \tilde{\psi}\left(D\left((xy+1)\delta(p_1(i), 7)\right), D\left((yz+2)\delta(p_2(i), 9)\right), p_3(i)\right)\Omega(\vec{x}, i) \\ &= \left(D\left((xy+1)\delta(p_1(i), 7)\right) + D\left((yz+2)\delta(p_2(i), 9)\right) + 1\right)\delta(p_3(i), 5)\Omega(\vec{x}, i).\end{aligned}$$

If $p_1(i) \neq 7$ or $p_2(i) \neq 9$ then Ω will annihilate the whole expression. And if $p_3(i) \neq 5$ then $\delta(p_3(i), 5)$ will annihilate the whole expresson. But assume $p_1(i) = 7$, $p_2(i) = 9$, and $p_3(i) = 5$. Then

$$\delta(p_1(i), 7) = \delta(p_2(i), 9) = \delta(p_3(i), 5) = \Omega(\vec{x}, i) = 1$$

and the whole expression becomes

$$\begin{aligned}\tilde{f}(\vec{x}, i) &= (D(xy+1) \cdot 1 + D(yz+2) \cdot 1 + 1) \cdot 1 \cdot 1 \\ &= D(xy+1) + D(yz+2) + 1 \\ &= xy + yz + 1 + 1 \\ &= f(\vec{x}) + 1.\end{aligned}$$

So $\tilde{f}$ is a gate to f, and explicitly its proper keys are those i such that, when i is factored into primes, it looks like $i = 2^7 3^9 5^5 \cdots$. Note that Ω is indispensible here. If we erased the Ω factor and evaluated $\tilde{f}(0, 0, 0, 5^5)$ we would get $D(1 \cdot 0) + D(2 \cdot 0) + 1 = 1 \neq f(0, 0, 0) + 1 = 2$ and so 5^5 would not be a key to f via $\tilde{f}$ at $(0, 0, 0)$, hence $\tilde{f}$ would not be a gate to f.

Lemma 3 demonstrates the notion of computation by guessing. In order to compute f, we take the gates $\tilde{\psi}, \tilde{\phi}_1, ..., \tilde{\phi}_r$ and *guess* proper keys for them. Most our guesses will be false, so we may have a long wait ahead! But if we're systematic, since proper keys do exist, eventually we will stumble upon them. It is relatively simple to "encode" this process formulaically in Lemma 3 (this type of encoding process is used often in computability theory and goes by the monicker of *Gödel numbering*); in the next two lemmas our methodology is similar but the details are more daunting.

Lemma 4: The set S in Lemma 3 is closed under recursion.

Proof: Suppose $g :\subseteq \mathbb{N}^{k-1} \to \mathbb{N}$, $h :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ are in S, so they have F_0 gates $\tilde{g} : \mathbb{N}^k \to \mathbb{N}$ and $\tilde{h} : \mathbb{N}^{k+2} \to \mathbb{N}$. Define $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ by the recurrence relation

$$f(\vec{n}, 0) = g(\vec{n})$$
$$f(\vec{n}, m+1) = h(\vec{n}, f(\vec{n}, m), m).$$

We must show f has an F_0 gate.

The process we encode is as follows: guess what $f(\vec{n}, 0), ..., f(\vec{n}, m-1)$ are and store these guesses. At this point we do not know whether these guesses are correct. To check, we can use the gates to g and h. Thus, guess a proper key for $\tilde{g}$ at $\vec{n}$ and guess proper keys for $\tilde{h}$ at $(\vec{n}, f(\vec{n}, 0), 0), ..., (\vec{n}, f(\vec{n}, m-1), m-1)$; store these guesses. We can proceed if and only if *all* these numerous guesses happened to be correct. If not, we shall throw them away and guess again until, eventually, we succeed! To check if a guess is correct, we need to check if all the guessed keys are proper, and then check if the guesses for the $f(\vec{n}, i)$ were correct by using the keys and the definition of f. A guess will be encoded in an integer, and a completely correct guess will correspond to a proper key to the new gate for f. Information will of course be stored in prime exponents. Here is our "f key blueprint":

$$\begin{aligned}
p_1(i) &: \text{A guess of a proper key for } h \text{ at } (\vec{n}, f(\vec{n}, 0), 0) \\
p_2(i) &: \text{A guess of } f(\vec{n}, 0) \\
p_3(i) &: \text{A guess of a proper key for } h \text{ at } (\vec{n}, f(\vec{n}, 1), 1) \\
p_4(i) &: \text{A guess of } f(\vec{n}, 1) \\
&\vdots \\
p_{2m-1}(i) &: \text{A guess of a proper key for } h \text{ at } (\vec{n}, f(\vec{n}, m-1), m-1) \\
p_{2m}(i) &: \text{A guess of } f(\vec{n}, m-1) \\
p_{2m+1}(i) &: \text{A guess of a proper key for } g \text{ at } (\vec{n})
\end{aligned}$$

We build $\tilde{f}$ in pieces, each piece in F_0.

The basic idea is to multiply several factors together to test the validity of a guessed key, according to the above key blueprint. The first crucial factor will be 1 if the subkeys encoded in the guessed key are proper at the appropriate points, and 0 otherwise. We will formally define this factor below: it will be $P_1(\vec{n}, m, i)$. The second crucial factor will be 1 if the subkeys encoded in i are consistent with the guessed values of f encoded in i; and 0 otherwise. This will be $P_2(\vec{n}, m, i)$ below. $P_1(\vec{n}, m, i)$ will be a factor in $P_2(\vec{n}, m, i)$ so that, together with some minor additional factors, we will obtain a product which will equal 1 if our guessed key meets the key blueprint specifications above, and 0 otherwise.

Define $P_1 : \mathbb{N}^{k+1} \to \mathbb{N}$ as follows: $P_1(\vec{n}, m, i) = 1$ if $m > 0$ and the guessed keys encoded in i, according to the above blueprint, are all proper, assuming the guessed values of f are correct; $P_1(i) = 0$ otherwise. More formally, $P_1(\vec{n}, m, i) = 1$ if and only if $m > 0$ and $\tilde{g}(\vec{n}, p_{2m+1}) > 0$ and for all $0 \leq j < m$, $\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+1}(i)) > 0$. To convince the reader $P_1(\vec{n}, m, i)$ is in F_0, we note that

$P_1(\vec{n}, m, i) = \overline{\delta}(0, m)\overline{\delta}(0, \tilde{g}(\vec{n}, p_{2m+1}(i)))$
$\delta\left(0, \sum_{j=0}^{D(m)} \delta(0, \tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+1}(i)))\right)$.

To see this, note that

$$\delta(0, \tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+1}(i)))$$

is nonzero precisely if $\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+1}(i)) = 0$, so the sum is nonzero precisely if one of the guessed keys for h at $(\vec{n}, p_{2j+2}(i), j)$ is non-proper (where p_{2j+2} is what we are guessing for $f(\vec{n}, j)$; for now we're ignoring whether the guess for $f(\vec{n}, j)$ is correct). So

$$\delta\left(0, \sum_{j=0}^{D(m)} \delta(0, \tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+1}(i)))\right)$$

is 0 precisely if one of the guessed keys for h is non-proper, 1 otherwise. The role of the other two factors is similar but simpler.

Second, define $P_2 : \mathbb{N}^{k+1} \to \mathbb{N}$ thus: $P_2(\vec{n}, m, i) = 1$ if $m > 0$ and i encodes a completely accurate set of guesses, that is (formally), $m > 0$ and $P_1(\vec{n}, m, i) = 1$ (so the proper key guesses are all on the mark), $p_2(i) = D(\tilde{g}(\vec{n}, p_{2m+1}(i)))$ (so the guess for $f(\vec{n}, 0)$ agrees with $g(\vec{n})$) and for each $0 < j < m$, $p_{2j+2}(i) = D(\tilde{h}(\vec{n}, p_{2j}(i), j-1, p_{2j+1}(i)))$ (so the guess for $f(\vec{n}, j)$ is determined according to the recurrence relation which defines f). We claim

$P_2(\vec{n}, m, i) = \overline{\delta}(0, m)P_1(\vec{n}, m, i)\delta(p_2(i), D(\tilde{g}(\vec{n}, p_{2m+1}(i))))$
$\delta\left(D(m), \sum_{j=0}^{D(D(m))} \overline{\delta}(1, m)\delta(p_{2j+4}(i), D(\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+3}(i))))\right)$.

To see this, assume $m > 1$. The other two cases are similar with some minor extra details thrown in. Then we can ignore the $\overline{\delta}(1, m)$. For a given j, $p_{2j+4}(i)$ is our encoded guess for $f(\vec{n}, j+1)$ (consult the key blueprint), $p_{2j+2}(i)$ is our encoded guess for $f(\vec{n}, j)$ and $p_{2j+3}(i)$ is our encoded guess for a proper key to h at $(\vec{n}, f(\vec{n}, j), j)$ via $\tilde{h}$. We can assume the guessed h-key is proper since otherwise P_1 will kill off everything. Thus

$$\delta(p_{2j+4}(i), D(\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+3}(i))))$$

is 0 precisely if $h(\vec{n}, p_{2j+2}, j) = p_{2j+4}$, or in English: this quantity is 0 if and only if , assuming our guess for $f(\vec{n}, j)$ is correct, then our guess for $f(\vec{n}, j+1)$ satisfies the recurrence relation f was defined by, and so our guess for $f(\vec{n}, j+1)$ is also correct. If our guess for $f(\vec{n}, 0)$ is incorrect then the factor

$$\delta(p_2(i), D(\tilde{g}(\vec{n}, p_{2m+1}(i))))$$

kills everything off (consult the key blueprint), so assume our guess for $f(\vec{n},0)$ is correct. Since we are assuming $m > 1$,

$$\sum_{j=0}^{D(D(m))} \overline{\delta}(1,m)\delta(p_{2j+4}(i), D(\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+3}(i))))$$

equals $D(m)$ precisely if each

$$\delta(p_{2j+4}(i), D(\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+3}(i))))$$

is 1, which by the above reasoning is true precisely if (since our guess for $f(\vec{n},0)$ is correct) our guess for $f(\vec{n},1)$ is correct and (since our guess for $f(\vec{n},1)$ is correct) our guess for $f(\vec{n},2)$ is correct, and so on. Ergo, when our guess at $f(\vec{n},0)$ and all our key-guesses are true, then

$$\sum_{j=0}^{D(D(m))} \overline{\delta}(1,m)\delta(p_{2j+4}(i), D(\tilde{h}(\vec{n}, p_{2j+2}(i), j, p_{2j+3}(i))))$$

is $D(m)$ precisely if all our guesses for $f(\vec{n},j)$ are correct; from here the claim is clear. Evidently $P_2 \in F_0$.

Finally, combine the pieces and define $\tilde{f} : \mathbb{N}^{k+1} \to \mathbb{N}$ by

$$\tilde{f}(\vec{n},m,i) = \delta(0,m)\tilde{g}(\vec{n},i) + P_2(\vec{n},m,i)\tilde{h}(\vec{n}, p_{2m}(i), D(m), p_{2m-1}(i)).$$

We claim that $\tilde{f}$ is a gate for f. If $m = 0$ then $\tilde{f}(\vec{n},m,i)$ is just $\tilde{g}(\vec{n},i)$ (P_2 vanishes since it is a multiple of $\overline{\delta}(0,m)$) and it inherits $\tilde{g}$'s keys exactly. Assume $m > 0$. The way we've designed it, $\tilde{f}(\vec{n},m,i)$ will be 0 if i does not fit the above blueprint perfectly. But if it does fit the blueprint (and at least one such i exists, assuming $\vec{n} \in dom\, f$, since $\tilde{g}$ and $\tilde{h}$ have proper keys), $P_2(\vec{n},m,i) = 1$ disappears from the product and

$$\begin{aligned}
\tilde{f}(\vec{n},m,i) &= \tilde{h}(\vec{n}, p_{2m}(i), D(m), p_{2m+1}(i)) \\
&= \tilde{h}(\vec{n}, f(\vec{n},m-1), m-1, p_{2m+1}(i)) \\
&= h(\vec{n}, f(\vec{n},m-1), m-1) + 1 \\
&= f(\vec{n},m) + 1.
\end{aligned}$$

So $\tilde{f}$ is the gate we desired, proving the lemma. ■

In the above proof, as an example of how the "key blueprint" works, the key $78750 = 2^1 \times 3^2 \times 5^4 \times 7^1$ corresponds to guessing that 1 is a proper key to h at $(\vec{n}, f(\vec{n},0), 0)$ via $\tilde{h}$, that $f(\vec{n},0) = 2$, that 4 is a proper key to h at $(\vec{n}, f(\vec{n},1), 1)$, that $f(\vec{n},1) = 1$, that $f(\vec{n},j) = 0$ for $j = 2,3,...,m-1$, that 0 is a proper key to h at $(\vec{n}, f(\vec{n},j))$ via $\tilde{h}$ (for $j = 2,3,...,m-1$), and that 0 is a proper key to g at $\vec{n}$.

The next lemma is much the same as the previous.

Lemma 5: Let S be as in Lemma 3. S is closed under unbounded minimization.

Proof: Let $g :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ be in S, and let $\tilde{g}$ be an F_0 gate for g. Define $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ in the usual way: $f(\vec{n})$ shall be the smallest m with $g(\vec{n}, m) = 0$ when such m exists (otherwise $\vec{n} \notin dom\, f$). We must show $f \in S$.

The procedure is almost identical to that in Lemma 4. We proceed by guessing what the answer is, then we guess keys to the gate for g and use them, and the definition of f, to make sure our guesses are correct. Guesses are stored as prime exponents in an integer, and an integer encoding a perfect sequence of guesses shall be a proper key to the (as yet unconstructed) gate $\tilde{f}$. Here is the "key blueprint":

$$
\begin{aligned}
p_1(i) &: \text{Guess for } f(\vec{n}), \text{ i.e., how far we need to search to find a zero} \\
p_2(i) &: \text{Guess for a proper key to } g \text{ at } (\vec{n}, 0) \text{ via } \tilde{g} \\
p_3(i) &: \text{Guess for a proper key to } g \text{ at } (\vec{n}, 1) \text{ via } \tilde{g} \\
&\vdots \\
p_{p_1(i)+2}(i) &: \text{Guess for a proper key to } g \text{ at } (\vec{n}, p_1(i)) \text{ via } \tilde{g}
\end{aligned}
$$

Once again, we build $\tilde{f}$ in pieces. First we define $P_1(\vec{n}, i)$ to equal 1 if the key-guesses encoded in i are entirely accurate (ignoring whether the guess for $f(\vec{n})$ is itself accurate for the time being). This is easy: we just check that $\tilde{g}(\vec{n}, j, p_{j+2}(i)) > 0$ for $0 \leq j \leq p_1(i)$. Thus note

$$P_1(\vec{n}, i) = \delta\left(0, \sum_{j=0}^{p_1(i)} \delta(0, \tilde{g}(\vec{n}, j, p_{j+2}(i)))\right).$$

Next we define $P_2(\vec{n}, i)$ to equal 1 if $\tilde{g}(\vec{n}, j, p_{j+2}(i)) \neq 1$ for all $0 \leq j < p_1(i)$; and 0 otherwise. Loosely speaking, if we assume the key-guesses are accurate, $P_2(\vec{n}, i)$ equals 1 if and only if $g(\vec{n}, j)$ is nonzero (or undefined) for all j up to (but not including) what we are guessing is $f(\vec{n})$. If $g(\vec{n}, j)$ *is* undefined for some j, there is no need to worry: $P_1(\vec{n}, i)$ will turn the entire product to 0, because it is impossible to guess a proper key for a point not even in the domain! Anyway, we note now

$$P_2(\vec{n}, i) = \delta\left(0, \sum_{j=0}^{p_1(i)} \overline{\delta}(j, p_1(i))\delta(1, \tilde{g}(\vec{n}, j, p_{j+2}(i)))\right).$$

We had to do some gymnastics here: the idea is to take the sum up to $p_1(i) - 1$, but troubles arise if $p_1(i) = 0$ since Definition 2 does not accomodate the symbol $\sum_0^{-1}$, so instead we sum all the way up to $p_1(i)$, but "turn off" the $p_1(i)$ term itself by multiplying by $\overline{\delta}(j, p_1(i))$.

Finally, define

$$\tilde{f}(\vec{n},i) = P_1(\vec{n},i)P_2(\vec{n},i)(p_1(i)+1)\delta(1,\tilde{g}(\vec{n},p_1(i),p_{p_1(i)+2}(i))).$$

We claim $\tilde{f}$ is a gate for f. The reasoning is very similar to that in Lemma 4 and we omit the details. $\tilde{f}$ certainly is in F_0 so this proves the lemma. ■

As promised, these lemmas allow us to give an alternate characterization of the computable functions. In honor of the methodology used in Lemmas 3, 4, and 5, we give this next lemma a name:

Lemma 6 (The Guessing Lemma): Let $f :\subseteq \mathbb{N}^k \to \mathbb{N}$. Then f is computable if and only if there exists a gate $\tilde{f}$ to f with $\tilde{f} \in F_0$.

Proof: Suppose $\tilde{f} \in F_0$ is a gate to f. Here is an algorithm to compute $f(\vec{n})$:

1. Take input $\vec{n} \in \mathbb{N}^k$.
2. Let $I = 0$.
3. If $\tilde{f}(\tilde{n},I) > 0$, output $\tilde{f}(\vec{n},I)-1$ and halt.
4. Add 1 to I and return to step 3.

Line 3 in the algorithm is effectively computable because all the machinery of Definition 2 is effectively computable, except for infinite series; but we do not need infinite series to deal with $\tilde{f}$ because it lies in F_0. By the Church Thesis, f is computable.

To prove the converse, we use induction on the number of steps needed to prove f is recursive using the statements of Definition 1. If f can be proven recursive in just one step, then f is O, σ, or π^k_j for some $1 \leq j \leq k$. These have F_0 gates $\tilde{O}(n,i) = 1$, $\tilde{\sigma}(n,i) = n+2$ and $\tilde{\pi}^k_j(\vec{n},i) = n_j + 1$, all evidently F_0. Now suppose f takes $k > 1$ steps to prove recursive using Definition 1. Then maybe f can be defined, using part 2 of Definition 1, as the composition of functions which can be proven computable in fewer than k steps each. Each of these functions, by induction, has an F_0 gate, and so by Lemma 3, so does f. Or maybe f can be defined, using part 3 of Definition 1, using recursion on functions which can be proven computable in fewer than k steps each. Each of these functions, by induction, has an F_0 gate, so by Lemma 4, f does. Finally, maybe f can be defined using unbounded minimization on a function that can be proven to be computable in fewer than k steps. Then again that function has an F_0 gate by induction, and by Lemma 5, so does f. ■

With the Guessing Lemma under our belt, we can state the main result of this section, which follows quite easily.

Theorem 7: Every computable function has a formula, in the sense of Definition 2.

Loosely speaking, every computable function has an explicit formula using the machinery from Definition 2.

Proof of Theorem 7: Let $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ be recursive. By the Guessing Lemma, there is a gate $\hat{f} : \mathbb{N}^{k+1} \to \mathbb{N}$, $\hat{f} \in F_0$. Let $D : \mathbb{N} \to \mathbb{N}$ be defined by $D(0) = 0$, $D(n) = n - 1$ for $n > 0$ (recall $D \in F_0$). We claim that

$$f(\vec{n}) = D\left(\hat{f}\left[\vec{n}, \sum_{i=0}^{\infty} \overline{\delta}(0, \hat{f}(\vec{n}, i))\delta\left(0, \sum_{j=0}^{i} \overline{\delta}(j, i)\overline{\delta}(0, \hat{f}(\vec{n}, j))\right)\right]\right),$$

where of course if $\vec{n} \notin dom\, f$ then both sides of the equation are undefined.

To see this, we evaluate from the inside out. The term $\overline{\delta}(j, i)\overline{\delta}(0, \hat{f}(\vec{n}, j))$ is 0 if $j = i$ or if j is not a proper key to f at $\vec{n}$ via $\hat{f}$. Therefore, the sum

$$\sum_{j=0}^{i} \overline{\delta}(j, i)\overline{\delta}(0, \hat{f}(\vec{n}, j))$$

is precisely the number of proper keys j such that $0 \leq j < i$. Knowing this, we see immediately that $\overline{\delta}(0, \hat{f}(\vec{n}, i))\delta\left(0, \sum_{j=0}^{i} \overline{\delta}(j, i)\overline{\delta}(0, \hat{f}(\vec{n}, j))\right)$ is 1 if i is not a proper key to f at $\vec{n}$ via $\hat{f}$ *and* there are no keys smaller than i. So that if $\vec{n} \in dom\, f$ then $\sum_{i=0}^{\infty} \overline{\delta}(0, \hat{f}(\vec{n}, i))\delta\left(0, \sum_{j=0}^{i} \overline{\delta}(j, i)\overline{\delta}(0, \hat{f}(\vec{n}, j))\right)$ is a sum where we keep adding 1 until the smallest proper key is found, and 0 after that. For example, if 3 is the smallest proper key, this sum becomes $1+1+1+0+0+0+\cdots$. So in fact this sum *is* the smallest proper key. Therefore the claim that the equation holds is immediate. On the other hand, if $\vec{n} \notin dom\, f$, then there is no proper key, so this formula for a proper key becomes $\sum_{i=0}^{\infty} 1$ which diverges, so both sides of the equation are undefined.

This proves the theorem, since we've written a formula for f. ■

The Guessing Lemma and Theorem 7 are constructive, given a proof that f is recursive (for examples of proofs of recursiveness, see the appendix). Informally, this basically says the hard problem of finding a formula for a function is reduced to the routine problem of programming that function in one's computer language of choice! Of course, if one follows these recipes directly, they result in some astonishingly complicated formulas.

Example: To illustrate the constructive aspect of the theory so far exposed, we will apply Theorem 7 to obtain a formula for the + function $+(n, m) = n + m$. Of course $+(n, m) = n + m$ is itself a formula, indeed a much simpler one than the one we will arrive at, so this excursion has no practical purpose.

We begin by giving a direct proof that + is recursive. This is needed because our theorem is constructive based on a proof of recursiveness. First, the projections π_1^1, π_2^3 and the incrementer σ are recursive by the base case in the definition of recursive. By closure under composition, $\sigma \circ \pi_2^3 : \mathbb{N}^2 \to \mathbb{N}$, given by $(\sigma \circ \pi_2^3)(a, b, c) = b + 1$, is recursive. Now we can define + by the recurrence

relation

$$\begin{aligned} +(n,0) &= \pi_1^1(n) \\ +(n,m+1) &= (\sigma \circ \pi_2^3)(n,+(n,m),m), \end{aligned}$$

so that $+$ is recursive by closure under recursion.

The Guessing Lemma yields the gates $\tilde{\pi}_1^1(n,i) = n+1$ for π_1^1, $\tilde{\pi}_2^3(a,b,c,i) = b+1$ for π_2^3, and $\tilde{\sigma}(n,i) = n+2$ for σ. The gate for $\sigma \circ \pi_2^3$ is more complicated to find: the Guessing Lemma proof tells us to consult Lemma 3, which provides the gate

$$\begin{aligned} \widetilde{(\sigma \circ \pi_2^3)}(a,b,c,i) &= \tilde{\sigma}\left(D(\tilde{\pi}_2^3(a,b,c,p_1(i))), p_2(i)\right)\delta(0,\delta(0,\tilde{\pi}_2^3(a,b,c,p_1(i)))) \\ &= (D(b+1)+2)\,\delta(0,\delta(0,b+1)) \\ &= (b+2)\delta(0,0) \;=\; b+2. \end{aligned}$$

Now all the easy work is done and we jump into finding the gate for $+$. The Guessing Lemma proof tells us to follow the instructions in Lemma 4. This is done by first defining P_1 and P_2 as so:

$$\begin{aligned} P_1(n,m,i) &= \overline{\delta}(0,m)\overline{\delta}(0,\tilde{\pi}_1^1(n,p_{2m+1}(i)))\delta\left(0,\textstyle\sum_{j=0}^{D(m)}\delta(0,\widetilde{(\sigma\circ\pi_2^3)}(n,\right. \\ &\qquad \left. p_{2j+2}(i),j,p_{2j+1}(i)))\right) \\ &= \overline{\delta}(0,m)\overline{\delta}(0,n+1)\delta\left(0,\textstyle\sum_{j=0}^{D(m)}\delta(0,p_{2j+2}(i)+2)\right) \\ &= \overline{\delta}(0,m)\delta\left(0,\textstyle\sum_{j=0}^{D(m)}0\right) \;=\; \overline{\delta}(0,m). \end{aligned}$$

Next we define P_2 by:

$$\begin{aligned} P_2(n,m,i) &= \overline{\delta}(0,m)P_1(n,m,i)\delta\left(p_2(i),D(\tilde{\pi}_1^1(n,p_{2m+1}(i)))\right)\delta\left(D(m),\right. \\ &\qquad \left.\textstyle\sum_{j=0}^{D(D(m))}\overline{\delta}(1,m)\delta(p_{2j+4}(i),D(\widetilde{(\sigma\circ\pi_2^3)}(n,p_{2j+2}(i),j,p_{2j+3}(i))))\right) \\ &= \overline{\delta}(0,m)\overline{\delta}(0,m)\delta(p_2(i),D(n+1))\delta\left(D(m),\textstyle\sum_{j=0}^{D(D(m))}\overline{\delta}(1,m)\right. \\ &\qquad \left.\delta(p_{2j+4}(i),D(p_{2j+2}(i)+2))\right) \\ &= \overline{\delta}(0,m)\delta(p_2(i),n)\delta\left(D(m),\textstyle\sum_{j=0}^{D(D(m))}\overline{\delta}(1,m)\right. \\ &\qquad \left.\delta(p_{2j+4}(i),p_{2j+2}(i)+1)\right). \end{aligned}$$

Putting these together, we obtain the gate

$$\begin{aligned} &\tilde{+}(n,m,i) \\ &= \delta(0,m)\tilde{\pi}_1^1(n,i) + \overline{\delta}(0,m)P_2(n,m,i)\widetilde{\sigma\circ\pi_2^3}(n,p_{2m}(i),D(m),p_{2m-1}(i)) \\ &= \delta(0,m)(n+1) + \overline{\delta}(0,m)P_2(n,m,i)(p_{2m}(i)+2) \\ &= \delta(0,m)(n+1) \\ &\quad + \overline{\delta}(0,m)\delta(p_2(i),n)\delta\left(D(m),\sum_{j=0}^{D(D(m))}\overline{\delta}(1,m)\delta(p_{2j+4}(i),p_{2j+2}(i)+1)\right)(p_{2m}(i)+2). \end{aligned}$$

Finally, using Theorem 7 with this gate, we get the formula

$$+(m,n) = D\left(\tilde{+}\left[n, m, \sum_{i=0}^{\infty} \overline{\delta}(0, \tilde{+}(n,m,i))\delta\left(0, \sum_{j=0}^{i} \overline{\delta}(j,i)\overline{\delta}(0, \tilde{+}(n,m,j))\right)\right]\right).$$

Expanding out the $\tilde{+}$'s in this formula is straightforward but tedious beyond measure, leading to a very large formula for $m+n$. Of course, a much simpler formula is just $+(m,n) = m+n$. The moral of the story is that in practice, it is far wiser to apply heuristics and ingenuity to obtain a formula for a function, rather than hash out the construction verbatim. $+$ is an extremely simple function, taking only a few steps to prove computable. More complicated functions which take more steps to prove computable lead to much larger formulas, if the construction is used directly. Nevertheless, it is philosophically fascinating that a construction does exist for obtaining formulas for computable functions, however impractical it might be.

It's natural to ask whether the converse to Theorem 7 is true. As a matter of fact, it is not. Indeed the converse fails very badly: it turns out even much more general functions have formulas using no more advanced machinery; the remainder of this section will elaborate on this claim.

The *arithmetical hierarchy* is a broad class of subsets of $\mathbb{N}$, about which quite a lot has been written, see for example [4]. The hierarchy consists of sets Π_k, Σ_k for each $k \in \mathbb{N}$. The elements of Π_k and Σ_k are certain subsets of natural numbers. Specifically: Σ_k is the set of all sets of the form

$$\{n | \exists m_1\, \forall m_2\, \exists m_3\, \cdots Q m_k\ f(n, m_1, ..., m_k) = 1\}$$

where the quantifiers start with $\exists$ and alternate thereafter (so $Q = \exists$ if k is odd, $Q = \forall$ if k is even), and where $f : \mathbb{N}^{k+1} \to \{0,1\}$ is computable. Π_k is the set of all sets of the form

$$\{n | \forall m_1\, \exists m_2\, \forall m_3\, \cdots Q m_k\ f(n, m_1, ..., m_k) = 1\};$$

here everything is as in the definition of Σ_k except that the alternating logic quantifiers begin with $\forall$ instead of $\exists$. Note that in particular when $k = 0$, the list of quantified m's is empty and we have $\Sigma_0 = \Pi_0$ is the set of sets of the form $\{n | f(n) = 1\}$ where $f : \mathbb{N} \to \{0,1\}$ is computable.

Note that each $\Sigma_i \subseteq \Sigma_{i+1}$ and each $\Pi_i \subseteq \Pi_{i+1}$. To see this, take some $f : \mathbb{N}^{k+1} \to \{0,1\}$ and note it can be extended to $f' : \mathbb{N}^{k+2} \to \mathbb{N}$ via $f'(n_1, ..., n_{k+2}) = f(n_1, ..., n_{k+1})$, i.e., ignoring the last input. Clearly f' will be computable if f was. Now the statement

$$\forall m_k\, \exists m_{k+1}\ f'(n, m_1, ..., m_{k+1}) = 1$$

is identical to the statement

$$\forall m_k\ f(n, m_1, ..., m_k) = 1.$$

Similar remarks hold if we swap $\exists$ and $\forall$. So f contributes the same set to Π_{k+1} or Σ_{k+1}, by way of f', that it contributes to Π_k or Σ_k. As a matter of fact, it's possible to show that the inclusions are proper, so the hierarchy really is a hierarchy as the name suggests.

Given a set $S \subseteq \mathbb{N}$, we define its *indicator function* $1_S : \mathbb{N} \to \{0,1\}$ by

$$1_S(n) = \begin{cases} 1 \text{ if } n \in S \\ 0 \text{ otherwise.} \end{cases}$$

The reason we care about the arithmetical hierarchy is that it turns out (see [4]) that many sets in the arithmetical hierarchy have noncomputable indicator functions.[2]

This is actually not surprising: given a set

$$S = \{n | \exists m \ f(n,m) = 1\}$$

from Σ_1, consider the problem of writing an algorithm to check whether a number n is contained in S. The brute force solution is to check the value of $f(n,m)$ for *every* $m \in \mathbb{N}$, or until one m were found with $f(n,m) = 1$. But if $n \notin S$, this algorithm will run forever, as its unfortunate implementor keeps checking more and more m. Maybe $f(n,m) = 0$ for the first trillion values of m, but we cannot rule out the possibility that the next value of m will finally have $f(n,m) = 1$! So brute force fails. If f is well-behaved (for example constantly 0), then an ad hoc alternative to brute force might work, but in general there will be functions f such that S has noncomputable indicator function. And of course, the daunting task of writing an algorithm becomes more and more hopeless as the number of logical quantifiers increases.

The computer scientists among the audience will note the arithmetical hierarchy includes sets whose indicator functions are minor variations of the Halting Function, which is the canonical example of a noncomputable function. Informally speaking, pick a computer programming language of choice. There is no effective way of determining if a given program in that language will halt when given its own code as input; this is the Halting Problem, see practically any textbook on computability theory. But there *is* an effective way of checking if a given program will halt, when fed its own self as input, in a given number of steps: just simulate the program for the given number of steps and note whether it halts in that time or not. So, since programs can be thought of as natural numbers (thanks to the fact they're really just long binary words), there's a *computable* function $f : \mathbb{N}^2 \to \{0,1\}$ such that

$$f(n_1, n_2) = \begin{cases} 1 \text{ if computer program } n_1 \text{ halts on input } n_1 \text{ within} \leq n_2 \text{ steps} \\ 0 \text{ otherwise.} \end{cases}$$

Then the set $S = \{n_1 | \exists n_2 \ f(n_1, n_2) = 1\}$ lies in Σ_1, hence in the arithmetical hierarchy. But S is exactly the set of programs which eventually halt when

[2] In fact, it can be shown that the sets with computable indicator functions are exactly $\Sigma_1 \cap \Pi_1$.

given their own source codes as input. So 1_S cannot be computable: that would solve the Halting Problem!

Theorem 8: If $f : \mathbb{N}^{k+1} \to \{0,1\}$ has a formula in the sense of Definition 2, and

$$S = \{n | \exists m_1 \forall m_2 \exists m_3 \cdots Q m_k \, f(n, m_1, ..., m_k) = 1\},$$
$$S' = \{n | \forall m_1 \exists m_2 \forall m_3 \cdots Q' m_k \, f(n, m_1, ..., m_k) = 1\},$$

then the indicator functions 1_S and $1_{S'}$ have formulas in the sense of Definition 2.

Proof: First, a remark: define $f_\exists : \mathbb{N}^k \to \{0,1\}$, $f_\forall : \mathbb{N}^k \to \{0,1\}$ by

$$f_\exists(n, m_1, ..., m_{k-1}) = \begin{cases} 1 \text{ if } \exists m_k \text{ s.t. } f(n, m_1, ..., m_k) = 1 \\ 0 \text{ otherwise,} \end{cases}$$
$$f_\forall(n, m_1, ..., m_{k-1}) = \begin{cases} 1 \text{ if } \forall m_k \; f(n, m_1, ..., m_k) = 1 \\ 0 \text{ otherwise.} \end{cases}$$

Then $f_\exists$ and $f_\forall$ also have formulas as in Definition 2. Indeed, if we let $g(n, m_1, ..., m_k) = |1 - f(n, m_1, ..., m_k)|$ (so that g certainly has a formula) then

$$f_\exists(n, m_1, ..., m_{k-1}) = \sum_{m_k=0}^{\infty} f(n, m_1, ..., m_k) \delta \left(1, \sum_{j=0}^{m_k} f(n, m_1, ..., m_{k-1}, j) \right)$$
$$f_\forall(n, m_1, ..., m_{k-1}) = \left| 1 - \sum_{m_k=0}^{\infty} g(n, m_1, ..., m_k) \delta \left(1, \sum_{j=0}^{m_k} g(n, m_1, ..., m_{k-1}, j) \right) \right|.$$

To verify the first formula, note the m_kth term is 0 unless m_k is minimal such that $f(n, m_1, ..., m_k) = 1$. The second formula is verified with similar reasoning, and the fact that the statement $\forall m_k \, f(n, m_1, ..., m_k) = 1$ is equivalent to the statement $\neg \exists m_k g(n, m_1, ..., m_k) = 1$.

Now to prove the theorem, we use induction on k. Assume the theorem holds for function which take fewer than $k+1$ inputs.

Define $\tilde{f} : \mathbb{N}^k \to \{0,1\}$ by

$$\tilde{f}(n, m_1, ..., m_{k-1}) = \begin{cases} 1 \text{ if } Q m_k \; f(n, m_1, ..., m_k) = 1 \text{ (remember } Q \text{ is } \exists \text{ or } \forall) \\ 0 \text{ otherwise.} \end{cases}$$

Since $\tilde{f}$ is either $f_\exists$ or $f_\forall$, by the remarks near the beginning of the proof, $\tilde{f}$ has a formula. If $Q = \exists$ then let $P = \forall$, otherwise let $P = \exists$. Then

$$S = \{n | \exists m_1 \, \forall m_2 \cdots P m_{k-1} \; \tilde{f}(n, m_1, ..., m_{k-1}) = 1\}.$$

This is trivial to see: just consider separately the cases $n \in S$ and $n \notin S$. But by the induction assumption, now 1_S has a formula.

Similar reasoning shows $1_{S'}$ has a formula as well.

As for the base step, observe that when $k = 0$ we have the degenerate case

$$S = S' = \{n | f(n) = 1\},$$

and so in fact $1_S = 1_{S'} = f$, so has a formula. So by induction, the theorem holds. ■

Corollary 9: If S is any set in the arithmetical hierarchy, then 1_S has a formula in the sense of Definition 2. In particular, the converse of Theorem 7 is false.

Proof: Suppose $S \in \Sigma_k$. So there is a computable $f : \mathbb{N}^{k+1} \to \{0, 1\}$ with

$$S = \{n | \exists m_1 \forall m_2 \exists m_3 \cdots Q m_k \, f(n, m_1, ..., m_k) = 1\}.$$

By Theorem 7, f has a formula in the sense of Definition 2, therefore by Theorem 8, so does 1_S. Similar reasoning goes for $f \in \Pi_k$. ■

One direction in which our eyes are turned presently is the search for a precise classification of functions which have formulas as in Definition 2. To the reader who has studied the arithmetical hierarchy, Corollary 9 says this class of functions is extremely vast!

5 Extension to Real-Valued Functions

Our results extend fairly easily to certain more general real-valued functions on all of $\mathbb{R}$. We would like to point out that there is a whole subfield of computability theory dealing specifically with this: computable real analysis. A caveat is that, perhaps counter to intuition, most schools of thought here hold that discontinuous functions are noncomputable [5]. This seems strange at first glance because it means some very predictable, well-behaved functions are noncomputable. For example, the function which equals 1 at 0 and 0 everywhere else, is considered noncomputable, even though it is extremely simple! On the other hand, suppose one were trying to compute such a function in a laboratory environment, where the input variable was something like temperature, force, or velocity. Determining whether the input were 0 or just something very close to 0 could be extremely difficult– in fact impossible. A lab scientist is constrained by imperfect precision, even in the best of conditions, and so in an experiment, the function described defies estimation (at least near 0).

Of course this does not mean every continuous function is computable. In fact, if one takes any function $\mathbb{N} \to \mathbb{N}$ which is noncomputable in the classical sense and extends it in any way to a continuous function, for example by "connecting the dots", the result will be a noncomputable but continuous real function.

Because of these subtleties, actually giving a definition of a computable real function (and justifying it to any reasonable degree) is beyond the scope of this paper. But we can surmount this difficulty and extend our own results without actually having to delve too deeply into computable analysis.

The next theorem essentially says: if a continuous function can be effectively computed to whatever accuracy you like, then it has a formula (if some extra machinery is added to what we have been using so far). This is somewhat obscured in the wording of the theorem because we have to talk about approximating real numbers using nothing but natural numbers! The confusion is further compounded by the fact that negative numbers are involved. Keep all this in mind when reading the exact wording of the theorem.

In the statement of the theorem, we use decimal digits. Of course, some numbers have more than one decimal digit representation. For example, $1 = 0.999....$ For this reason, we insist that in the below theorem, any time there are two decimal representations to choose from, the one *without* an infinite tail of 9's must be chosen.

Theorem 10: Let $f : \mathbb{R} \to \mathbb{R}$ be continuous. Define the function $g : \mathbb{N}^5 \to \mathbb{N}$ in the following way. For $a, ..., e \in \mathbb{N}$, $g(a, b, c, d, e)$ is obtained as follows:

1. Divide $(-1)^c a$ by 10^b. Thus, a gives a set of digits, b specifies where to put the decimal point, and c specifies a sign.

2. Let $r = f((-1)^c a/10^b)$; note r is a real number, maybe not an integer!

3. From r, discard all digits which are more than d places to the right of the decimal point, so that there are exactly d digits kept after the decimal point.

4. Multiply the result by 10^d and call the *integer* which results k. (Multiplying by 10^d loosely means "erase the decimal point")

5. If $e = 0$ and $k \geq 0$ then $g(a, b, c, d, e)$ will be k. If $e = 1$ and $k < 0$ then $g(a, b, c, d, e)$ will be $-k$. In any other case, $g(a, b, c, d, e)$ will be 0. Loosely speaking, inputing $e = 0$ says, "only return the result if it is positive", and inputing $e = 1$ says, "only return the result if it is negative, and if so, return its absolute value".

Suppose that g is computable. Then there is a formula for f using the machinery of Definition 2 together with subtraction, division, the floor function, and the absolute value function.

Note: To be utterly pedantic, one ought to formally define "functions which have formulas using machinery of Definition 2 together with subtraction, division, the floor function, and the absolute value function", in a way similar to Definition 2. We omit the uninsightful details.

Proof: First, fix a,b,c and d in $\mathbb{N}$. We claim that, with the approximation correct up to at least d digits after the decimal point,

$$f\left((-1)^c \frac{a}{10^b}\right) \approx \frac{g(a, b, c, d, 0) - g(a, b, c, d, 1)}{10^d}.$$

Suppose $f((-1)^c a/10^b) \geq 0$. Then $g(a, b, c, d, 1)$ vanishes and $g(a, b, c, d, 0)$, by definition, is $f((-1)^c a/10^b)$, correct to d digits after the decimal point, times 10^d, so when we divide by 10^d we get exactly $f((-1)^c a/10^b)$ approximated to d digits after the decimal point. If $f((-1)^c a/10^b) < 0$, the reasoning is very similar.

For fixed a, b, c, we compute $f((-1)^c a/10^b)$ by first computing it to 0 digits past the decimal point, then to 1 digit past, then to 2, and so on. This will of course converge to $f((-1)^c a/10^b)$. But we do not have limits at our disposal (we have not allowed ourselves to use them in our formulas). So instead we'll simulate a limit of a sequence by first adding the first term, then adding the second term minus the first, then the third minus the second, and so on. For notational convenience, let $g_{abc}(d, e)$ denote $g(a, b, c, d, e)$. Then, for fixed $a, b, c \in \mathbb{N}$ and by the above remarks:

$$f\left((-1)^c \frac{a}{10^b}\right) = g_{abc}(0,0) - g_{abc}(0,1) + \sum_{i=0}^{\infty} \left[\frac{g_{abc}(i+1,0) - g_{abc}(i+1,1)}{10^{i+1}} - \frac{g_{abc}(i,0) - g_{abc}(i,1)}{10^i} \right].$$

Since we are assuming g is computable, by Theorem 7, in the above formula for $f((-1)^c a/10^b)$, we can replace each g with a subformula using just machinery of Definition 2. So there exists a formula for for $f((-1)^c a/10^b)$ using just machinery of Definition 2 along with division and subtraction.

Finally, for $r \in \mathbb{R}$, to compute $f(r)$ we'll approximate r accurately to 0 digits after the decimal point and compute f there; then repeat with an approximation accurate up to 1 digit after the decimal point in r; then with 2; etc. *This works because we are assuming f is continuous.* Generally, a formula which approximates r validly up to i places after the decimal point is

$$h(r, i) = (-1)^{1+\delta(r,|r|)} \frac{\lfloor 10^i |r| \rfloor}{10^i}.$$

One should think about this pictorially. Imagine holding the decimal point stationary and shifting r to the left by i digits. That gives $10^i r$. Now we want to chop off everything past the decimal point, and shift back to the right i digits by dividing by 10^i. The chopping can be effected by taking the floor function– assuming r is nonnegative. To generalize to arbitrary r, do the above process to $|r|$ and then fix the sign by multiplying by $(-1)^{1+\delta(r,|r|)}$.

Thus, again using an infinite sum to simulate a limit,

$$f(r) = f(h(r,0)) + \sum_{i=0}^{\infty} (f(h(r, i+1)) - f(h(r, i))).$$

Our remarks about $f((-1)^c a/10^b)$ apply here, so we can transform this formula into one using just machinery of Definition 2 along with division, subtraction, the floor function, and the absolute value function. ■

The theorem provides some insight into the laboratory dilemma described above. Say $f(x) = 1$ when $x = 0$, $f(x) = 0$ elsewhere. Now suppose a scientist is trying to evaluate f of the temperature (in Celsius) of an object. By misfortune, the temperature is 0, but an initial reading only establishes it is within .1 degree of 0. If we use the strategy in the theorem, it amounts to advising the scientist:

keep recalibrating the instruments to make more and more accurate measurements, until the resulting estimation of f is as good as you like. Unfortunately, since f is discontinuous at 0, it does not matter how well the instruments are calibrated. The scientist will *never* be able to get the maximum error of her estimate smaller than 1.

To close this section, we'll point out that it would not be tremendously difficult, using the ideas above, to generalize things to functions which take multiple real arguments.

6 Conclusion

We set out originally to explore the problem of obtaining closed formulas for classical computable functions. In order to attack this problem, it was necessary first to formally explain what exactly this means. Having done this, we were able to give a constructive proof that the computable functions have closed formulas. This effort took us on a detour through the elements of computability theory, and through a new formula-related characterization (the Guessing Lemma) of the computable functions which we proved equivalent to the usual definition.

We then proceeded above and beyond the original goals and outlined how the results can be extended to more general real-valued computable functions, and, in a different direction, to functions that are not computable but are at least within the arithmetical hierarchy.

Appendix: Proving a Function is Recursive

The purpose of this appendix is to provide, for the reader new to recursive function theory, some insight into the Church-Turing thesis.

The Church-Turing thesis informally says that the primitive recursive functions are precisely the functions for which there exist algorithms; or, alternatively, the functions that can be programmed onto an ideal computer. In computer science, computability theory is usually introduced via the Turing machine, a model of an ideal computer. This approach has the advantage that (perhaps after experimenting for awhile with the flexibility of the Turing machine) it is "obvious" that Turing computable functions are exactly the functions programmable onto a computer. One then proves that the Turing-computable functions are precisely the recursive functions.

Unfortunately, a comprehensive introduction to Turing machines, followed by a proof of equivalence with recursive functions, would take us miles too far afield. Excellent treatments can be found in abundance, for example in Bilaniuk [2] which is available for free online. In the meantime, we will console the reader with some examples that should at least make the Church-Turing thesis sound reasonable, if not clear. The reader will bear in mind that these examples typically would show up as exercises in introductory computability texts.

For the reader's convenience, here is the definition of the recursive functions, repeated from the background section:

Definition: The *computable* functions (also called the *recursive* functions) are defined inductively as follows:

1. The *basic* functions are recursive. These are the *zero function* $O : \mathbb{N} \to \mathbb{N}$ defined by $O(n) = 0$; the *successor function* $\sigma : \mathbb{N} \to \mathbb{N}$ defined by $\sigma(n) = n + 1$; and the *projection functions* defined, for all $i, k \in \mathbb{N}$, $0 < i \leq k$, by

$$\pi_i^k : \mathbb{N}^k \to \mathbb{N} : (n_1, ..., n_k) \mapsto n_i.$$

2. (Closure under Composition) If $\psi :\subseteq \mathbb{N}^k \to \mathbb{N}$ is recursive and for $1 \leq i \leq k$, $\varphi_i :\subseteq \mathbb{N}^m \to \mathbb{N}$ is recursive, then the function $f :\subseteq \mathbb{N}^m \to \mathbb{N}$ defined by

$$f(\vec{n}) = \psi(\varphi_1(\vec{n}), ..., \varphi_k(\vec{n}))$$

   is also recursive. Here we write $\vec{n}$ for $n_1, ..., n_m$; this will be done frequently throughout the paper, and will cause no confusion.

3. (Closure under Recursion) If $g :\subseteq \mathbb{N}^k \to \mathbb{N}$ is recursive and $h :\subseteq \mathbb{N}^{k+2} \to \mathbb{N}$ is recursive, then the function $f :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ defined by the recurrence relation

$$f(\vec{n}, m) = \begin{cases} g(\vec{n}) \text{ if } m = 0, \\ h(\vec{n}, f(\vec{n}, m-1), m-1) \text{ otherwise} \end{cases}$$

   is recursive.

4. (Closure under Unbounded Minimization) Suppose $g :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ is recursive. Define the function $f :\subseteq \mathbb{N}^k \to \mathbb{N}$ by

$$f(x_1, ..., x_k) = \min\{y | g(x_1, ..., x_k, y) = 0 \text{ and } g(x_1, ..., x_k, z) \text{ exists } \forall z < y\},$$

   whenever any such y exists. Then f is recursive.

Now to give some examples. We start by showing how addition, multiplication, and exponentiation are proven to be recursive.

Example A1: The functions $+(a, b) = a + b$, $\times(a, b) = ab$, and $\exp(a, b) = a^b$ are recursive (where we define 0^0 to be 1).

Proof: First, observe that $+$, $\times$, and exp can be defined by the following recurrence relations:

$$\begin{aligned} +(a, 0) &= a \\ +(a, b+1) &= +(a, b) + 1 \\ \times(a, 0) &= 0 \\ \times(a, b+1) &= a + \times(a, b) \\ \exp(a, 0) &= 1 \\ \exp(a, b+1) &= \exp(a, b) \cdot a. \end{aligned}$$

To prove $+$ is recursive, note that $\pi_1^1(a) = a$, $\sigma(a) = a + 1$, and $\pi_2^3(a, b, c) = b$ are recursive, by definition. We can compose σ and π_2^3 to see that $(\sigma \circ \pi_2^3)(a, b, c) = b + 1$ is recursive. Now we can rewrite the recurrence relation for $+$ as follows:

$$\begin{aligned} +(a, 0) &= \pi_1^1(a) \\ +(a, b + 1) &= (\sigma \circ \pi_2^3)(a, +(a, b), b). \end{aligned}$$

Why on earth do we do this, when it seems to just complicate matters? Because this is exactly the format we need to apply closure under recursion! By closure under recursion, $+$ is indeed recursive.

Now define $+_3(a, b, c) = a + b$. We claim $+_3$ is recursive. This is just a matter of rewriting it as $+_3(a, b, c) = +(\pi_1^3(a, b, c), \pi_2^3(a, b, c))$. We just showed $+$ is recursive, and π_1^3, π_2^3 are recursive by definition, so indeed $+_3$ is recursive. Now we can use $+_3$, and the basic function $O(a) = 0$, to rewrite the recurrence relation for $\times$:

$$\begin{aligned} \times(a, 0) &= O(a) \\ \times(a, b + 1) &= +_3(a, \times(a, b), b). \end{aligned}$$

Again, at first glance this just muddies the waters, but looking at the definition of recursive functions, we see this is exactly the form we need to apply closure under recursion. By closure under recursion, $\times$ is recursive.

Continuing in this manner, define $\times_3(a, b, c) = ab$. Write $\times_3(a, b, c) = \times(\pi_1^3(a, b, c), \pi_2^3(a, b, c))$ and conclude by closure under composition that $\times_3$ is recursive. Note the constantly 1 function $\mathbf{1}(a) = 1$ is recursive by writing $\mathbf{1}(a) = \sigma(O(a))$. So, we can write

$$\begin{aligned} \exp(a, 0) &= \mathbf{1}(a) \\ \exp(a, b + 1) &= \times_3(a, \exp(a, b), b), \end{aligned}$$

and by closure under recursion we realize exp is recursive. ■

What the above example demonstrates is that a lot of the work in these types of proof is just a lot of acrobatics to make sure dimensions agree and to squeeze various recurrence relations into the specific format required for closure under recursion. The audience may wonder why we do not just save ourselves the trouble and use a stronger closure under recursion statement for the definition of recursive functions. This would indeed simplify the task of proving things are recursive, but it would greatly hinder us from proving things about recursive functions. For example, before being able to prove the Guessing Lemma, we would have to prove a stronger version of Lemma 4. We forfeit some ease of proving recursiveness in exchange for an easier time proving general theorems about recursive functions. In fact, if we are willing to invoke the Church-Turing thesis to prove things are recursive, we have not forfeited anything at all and have, by a kind of sleight of hand, drastically reduced our workload[3]!

[3]See the remarks in the section, "Background"

We showed + is recursive, so it is natural to want to show $-(a,b) = a - b$ is recursive. But this is not true! The reason is that recursive functions have to have strictly nonnegative values. We do the next best thing and show $-(a,b) = |a-b|$ is recursive.

Example A2: The "absolute value subtraction function" $-(a,b) = |a-b|$ is recursive.

Proof: This is actually not as easy as one would think; one cannot immediately mimic Example A1. We begin by proving

$$\text{SubtractOne}(a) = \begin{cases} a-1 \text{ if } a > 0 \\ 0 \text{ if } a = 0 \end{cases}$$

is recursive. To this end, define $\text{SubtractOne}'(a,b) = \text{SubtractOne}(b)$. Now why on earth do we pull this out of thin air? Because it turns out to be easy to get a recurrence relation for SubtractOne$'$. Namely:

$$\begin{aligned} \text{SubtractOne}'(a,0) &= 0 \\ \text{SubtractOne}'(a,b+1) &= \pi_3^3(a, \text{SubtractOne}'(a,b), b). \end{aligned}$$

The reader should take a moment to convince herself this is true. By closure under recursion, SubtractOne$'$ is recursive. But we can immediately write $\text{SubtractOne}(b) = \text{SubtractOne}'(\pi_1^1(b), \pi_1^1(b))$, so by closure under composition, SubtractOne is recursive. Now define

$$\text{HalfSubtract}(a,b) = \begin{cases} a-b \text{ if } a > b \\ 0 \text{ otherwise.} \end{cases}$$

Why pull this function out of a hat? Because clearly $-(a,b) = \text{HalfSubtract}(a,b) + \text{HalfSubtract}(b,a)$, so if we can show HalfSubtract is recursive, we're practically out of the woods. But this is actually easy, just write

$$\begin{aligned} \text{HalfSubtract}(a,0) &= a \\ \text{HalfSubtract}(a,b+1) &= (\text{SubtractOne} \circ \pi_2^3)(a, \text{HalfSubtract}(a,b), b), \end{aligned}$$

which in plain English is, "to half-subtract $b+1$, first half-subtract b, then half-subtract 1". By closure under recursion, HalfSubtract is recursive. Then since $-(a,b) = +(\text{HalfSubtract}(a,b), \text{HalfSubtract}(\pi_2^2(a,b), \pi_1^2(a,b)))$, the absolute value subtraction function is indeed recursive. ■

We are trying to give an idea how recursive functions encapsulate the intuitive notion of functions that can be programmed into a computer. Well, most decent programming languages should have a way of checking if some statement is true or false, and acting accordingly. The recursive analog is:

Example A3: The Kronecker delta function

$$\delta(a,b) = \begin{cases} 1 \text{ if } a = b \\ 0 \text{ otherwise} \end{cases}$$

is recursive.

Proof: Note that $\delta(a,b) = \delta(|a-b|, 0)$. Thus, define

$$\text{IsZero}(a) = \begin{cases} 1 \text{ if } a = 0 \\ 0 \text{ otherwise.} \end{cases}$$

We claim IsZero is recursive. Recall from the proof of example A2 that $\text{SubtractOne}(a) = a - 1$ if and only if $a = 0$, otherwise it equals a. So the difference between a and $\text{SubtractOne}(a)$ is 1 if a is nonzero, 0 if $a = 0$. Therefore, $\text{IsZero}(a) = |1 - |a - \text{SubtractOne}(a)||$. Or, in the language of the definition of recursive functions,

$$\text{IsZero}(a) = -(\mathbf{1}(a), -(a, \text{SubtractOne}(a))).$$

By closure under composition repeatedly, IsZero is recursive. But then immediately $\delta(a,b) = \text{IsZero}(-(a,b))$ is recursive.

Incidentally, there is another, more amusing proof that IsZero is recursive: remember in Example A1 we specifically defined $0^0 = 1$. But for any $n > 0$, $0^n = 0$. So in fact $\text{IsZero}(n) = \exp(0, n) = \exp(O(n), \pi_1^1(n))$.

Once we have that IsZero is recursive, the recursiveness of δ follows easily using the above observation. ■

Another feature of many good programming languages is the "FOR" loop. An analog of a for loop is the following.

Example A4: Suppose $f :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ and $g :\subseteq \mathbb{N}^k \to \mathbb{N}$ are recursive, and define $h :\subseteq \mathbb{N}^k \to \mathbb{N}$ by $h(\vec{n}) = \sum_{i=0}^{g(\vec{n})} f(\vec{n}, i)$. Then h is recursive.

Proof: Define $\text{FirstFewTerms} :\subseteq \mathbb{N}^{k+1} \to \mathbb{N}$ by $\text{FirstFewTerms}(\vec{n}, m) = \sum_{i=0}^{m-1} f(\vec{n}, i)$. We claim FirstFewTerms is recursive. The recurrence relation we will use to prove this is

$$\begin{aligned} \text{FirstFewTerms}(\vec{n}, 0) &= 0 \\ \text{FirstFewTerms}(\vec{n}, m+1) &= f(\vec{n}, m) + \text{FirstFewTerms}(\vec{n}, m). \end{aligned}$$

But if the skeptics insist we put this precisely in the form needed for closure under recursion, then we first define $\text{tmp}(\vec{n}, b, c) = b + f(\vec{n}, c)$. tmp is recursive because we can write it

$$\text{tmp}(\vec{n}, b, c) = +(\pi_{k+1}^{k+2}(\vec{n}, b, c), f(\pi_1^{k+2}(\vec{n}, b, c), ..., \pi_k^{k+2}(\vec{n}, b, c), \pi_{k+2}^{k+2}(\vec{n}, b, c)))$$

and apply closure under composition repeatedly. Then we satisfy the skeptics by writing

$$\begin{aligned} \text{FirstFewTerms}(\vec{n}, 0) &= f(\pi_1^k(\vec{n}), ..., \pi_k^k(\vec{n}, (O \circ \pi_1^k)(\vec{n}))) \\ \text{FirstFewTerms}(\vec{n}, m+1) &= \text{tmp}(\vec{n}, \text{FirstFewTerms}(\vec{n}, m), m). \end{aligned}$$

So FirstFewTerms is recursive. Now we have $h(\vec{n}) = \text{FirstFewTerms}(\vec{n}, g(\vec{n}))$. This should suffice to convince anyone that h is recursive, but it is worth pointing

out that if one wanted to be thoroughly pedantic, one would write

$$h(\vec{n}) = \text{FirstFewTerms}(\pi_1^k(\vec{n}), ..., \pi_k^k(\vec{n}), g(\vec{n}))$$

before appealing to closure under composition. ■

We would like to wrap the appendix up by showing that the nth prime function is recursive. This is a noble goal because it will provide an example of usage of closure under unbounded minimization, which until now we have not used. First, we need to show a couple of simpler functions are recursive:

Example A5: Define

$$\text{IsPrime}(n) = \begin{cases} 1 \text{ if } n \text{ is prime} \\ 0 \text{ otherwise.} \end{cases}$$

Define PrimeCount(n) to be the number of primes between 0 and n inclusive. IsPrime and PrimeCount are recursive.

Proof: First we prove IsPrime is recursive. How do we check if a number is prime? One way is to count the number of divisors. Thus, as an intermediate step, define

$$\text{ADividesB}(a, b) = \begin{cases} 1 \text{ if } a \text{ divides } b \\ 0 \text{ otherwise.} \end{cases}$$

We claim ADividesB is recursive. One way to check if a divides b is to let i go from 0 to b and at each step add 1 if $ai = b$. Note we add 1 at most once. A little more formally, we have $\text{ADividesB}(a, b) = \sum_{i=0}^{b} \delta(ai, b)$. It follows from example A4 that ADividesB is recursive. Now, n is prime if and only if the number of divisors of n between 0 and n inclusive is 2: 1 is a divisor, n is a divisor, and there are no other divisors. Thus, we have $\text{IsPrime}(n) = \delta\left(2, \sum_{i=0}^{n} \text{ADividesB}(i, n)\right)$. Again the recursiveness of IsPrime follows from example A4. Finally, write $\text{PrimeCount}(n) = \sum_{i=0}^{n} \text{IsPrime}(i)$. Once again, example A4 implies PrimeCount is recursive. ■

Example A6: Define $P(n)$ to be the nth prime (thus $P(0)$ is undefined). P is recursive.

Proof: We will now show how to use closure under unbounded minimization to simulate a "brute force search". One way to find the nth prime is to just start checking every single number i, in order, until we find the first i such that there are n primes between 0 and i inclusive. In other words, find the lowest i such that $\text{PrimeCount}(i) = n$. But this is not the format that closure under unbounded minimization requires. To get things in that format, define

$$\text{IsNotNthPrime}(n, i) = \begin{cases} 1 \text{ if } i \text{ is not the } n\text{th prime} \\ 0 \text{ otherwise} \end{cases}$$

It is not hard to see $\text{IsNotNthPrime}(n, i) = |1 - \text{IsPrime}(i)\delta(n, \text{PrimeCount}(i))|$. So by liberally invoking prior examples and the definition of recursive, IsNotNthPrime

is recursive. Now in terms of IsNotNthPrime, our quest is to find the smallest i such that IsNotNthPrime$(n, i) = 0$. But this is *exactly* what closure under unbounded minimization allows us to do! Applying closure under unbounded minimization to IsNotNthPrime, P is recursive. ■

Note what happens when we try to use unbounded minimization to find $P(0)$ the undefined "0th prime". We start checking numbers, asking each time, "Is it not the case that this is not the 0th prime?", i.e., "Is this the 0th prime"? We stop when we get a positive answer... but there *is* no 0th prime, so we never get a positive answer, and keep trying forever.

References

[1] H. P. Barendregt. *The lambda calculus*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 1984.

[2] Stefan Bilaniuk. *A problem course in mathematical logic*. 2003. Freeware mathematics text, Version 1.6, available at http://euclid.trentu.ca/math/sb/pcml/welcome.html.

[3] Nigel Cutland. *Computability*. Cambridge University Press, Cambridge, 1980. An introduction to recursive function theory.

[4] Hartley Rogers, Jr. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, MA, second edition, 1987.

[5] Klaus Weihrauch. *Computable analysis*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2000.